Haaga-Helia
ammattikorkeakoulu Oy

Reaktiivinen ohjelmointi Scalalla

Aleksi Tilli

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2017

Haaga-Helia
ammattikorkeakoulu Oy

Tiivistelmä

Tekijä(t)
Aleksi Tilli

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Raportin/Opinnäytetyön nimi
Reaktiivinen ohjelmointi Scalalla

Sivu- ja liitesivumäärä
35 + 0

Reaktiivinen ohjelmointi on ohjelmointiparadigma, joka on tapahtumapohjaisen ja deklaratiivisen ohjelmoinnin alalaji. Opinnäytetyössä perehdyttiin reaktiiviseen ohjelmointiin ja siihen liittyviin kirjastoihin Scala-ohjelmointikielellä. Kirjastoista selvitettiin mitä reaktiivisen ohjelmoinnin piirteitä ne toteuttavat, ja mihin reaktiivisen ohjelmoinnin alalajiin ne kuuluvat. Lisäksi valittiin lähempään tarkasteluun kirjastot, joilla toteutettiin esimerkkisovellus. Esimerkkisovelluksesta toteutettiin kaksi versiota, toinen Scalan futuureilla ja toinen kirjastoilla. Toteutusten koodirivien lukumäärää verrattiin, ja näin pyrittiin saamaan selville kirjastojen vaikutus koodin luettavuuteen.

Opinnäytetyössä käsiteltyjä kirjastoja olivat Akka Streams, Apache Spark Streaming, Reactive Extensions, Reactor, Sodium, Vert.x Rx ja Vert.x Sync. Esimerkkisovellusta varten lähempään tarkasteluun valittiin näistä Reactive Extensions ja Akka Streams.

Tuloksiksi saatiin, että esimerkkisovelluksessa reaktiivisen ohjelmoinnin kirjastoilla toteutetun version koodirivien lukumäärä oli suurempi kuin futuureilla toteutetussa versiossa. Lisäksi käsitellyt kirjastot olivat pääasiassa toistaiseksi luokittelemattomaan reaktiiviseen ohjelmointiin kuuluvia.

Asiasanat
ohjelmointi, paradigmat, reaktiivisuus

Sisällys

1 Johdanto

Tutkimuksessa perehdytään reaktiiviseen ohjelmointiin ja reaktiivisen ohjelmoinnin kirjastoihin Scala-ohjelmointikielellä. Opinnäytetyön motiivina on arvioida reaktiivisen ohjelmoinnin ja reaktiivisen ohjelmoinnin kirjastojen hyödynnettävyyttä työelämää varten. Keskeisinä tarkasteltavina asioina ovat kirjastojen käyttö ja vaikutus koodin luettavuuteen, sekä niiden toteuttamat reaktiivisen ohjelmoinnin piirteet. Tutkimus suoritetaan Scalalla, mutta tulokset ovat hyödynnettävissä myös muilla JVM-kielillä, kuten Javalla ja Clojurella. Koodiesimerkit on kirjoitettu Scalalla, mutta esitellyt kirjastot ovat saatavissa Javalle.

Aiheen ajankohtaisuudesta kertoo, että Scalaa ja useita JVM-ratkaisuja (Akka, Play-ohjelmistokehys) kehittävä organisaatio on saanut kasvavissa määrin kyselyitä asiasta vuonna 2015 ja erityisesti vuonna 2016. Siihen liittyy kuitenkin käsitteenä yhteisymmärryksen puute. (Bonér & Klang, 2016.) Funktionaalisen reaktiivisen ohjelmoinnin luoja Conal Elliott (Elliott, 13.6.2015) on havainnut vastaavanlaisia ongelmia. Funktionaalinen reaktiivinen ohjelmointi on kasvattanut suosiota, mutta tullut väärin ymmärretyksi viime vuosina. Lisäksi ajankohtaisuutta lisää reaktiiviseen ohjelmointiin liittyvän, vuonna 2015 julkaistun, Reactive Streams -standardin tuleminen osaksi Java 9 -standardia (Reactive Streams Special Interest Group, 2015; Oracle, 2017).

Tutkimus sai alkunsa käytännön ongelmasta työelämässä. Eräs Slick-tietokantakirjasto tuli päivittää uuteen versioon osana Scala-versiopäivitystä. Kirjaston käyttö oli kuitenkin muuttunut huomattavasti aiemmista versioista, mikä teki päivittämisestä työlästä. Aiemmasta poiketen kyseistä versiota markkinoitiin funktionaaliseksi ja reaktiiviseksi (Reactive Functional Relational Mapping For Scala). Tämä aiheutti sekaannusta useiden käsitteiden kanssa, kuten funktionaalinen reaktiivinen ohjelmointi, reaktiivinen ohjelma ja reaktiivisen manifestin (Reactive Manifesto) mukainen reaktiivinen järjestelmä (Reactive System), jota Scalaa ja Slick-kirjastoa kehittävä Lightbend markkinoi tuotteissaan. Tutkimuksen taustalla oleva merkittävä tekijä olikin ajatus siitä, että näiden, ja erityisesti reaktiivisen ohjelmoinnin syvempi ymmärtäminen olisi saattanut helpottaa kirjaston käyttöä.

Tutkimuksen tarkoituksena on syventää tekijän tietämystä Scalasta, ohjelmointiparadigmoista ja erityisesti teoreettista tietämystä reaktiivisesta ohjelmoinnista, jotta tekniikoiden arvioiminen tulevaisuudessa olisi helpompaa. Tutkimuksesta voi olla kuitenkin hyötyä myös muille Scalasta ja reaktiivisesta ohjelmoinnista kiinnostuneille.

1.1 Käsitteitä

Deklaratiivinen ohjelmointi

Ohjelmointiparadigma (kts. ohjelmointiparadigma), jossa ilmaistaan mitä tehdään, ja annetaan kielen hoitaa automaattisesti, miten se tehdään vrt. imperatiivinen ohjelmointi. Tätä noudattaa esimerkiksi SQL.

FRP

Lyhenne sanoista functional reactive programming, eli funktionaalinen reaktiivinen ohjelmointi.

Imperatiivinen ohjelmointi

Ohjelmointiparadigma (kts. ohjelmointiparadigma), jossa keskitytään ilmaisemaan ohjelman toiminta askel askeleelta. Vastaa kysymykseen, miten ohjelma suoritetaan. Tähän kuuluvat mm. assembly- ja C-ohjelmointikieli.

JVM

Lyhenne sanoista Java Virtual Machine, eli alustasta, jolla kieliä kuten Javaa, Scalaa ja Groovya ajetaan.

Kirjasto

Ohjelmoinnissa käytettävä uudelleen hyödynnettävä komponentti, jota voidaan käyttää useissa ohjelmissa.

Ohjelmistokehys (software framework)

Kirjastoa laajempi kokonaisuus, joka ohjaa ohjelmoijaa ohjelmoimaan ohjelma tietyllä tavalla.

Ohjelmointiparadigma

Tapa ajatella, ja ilmaista ratkaisu ohjelmointiongelmaan

Säie (thread)

Prosessi, eli ohjelma koostuu vähintään yhdestä kevyemmästä ohjelmasta, säikeestä, jossa komentoja suoritetaan.

Tila (state)

Ohjelmassa käytettyjen muuttujien arvot, joiden muuttuessa myös ohjelman tila muuttuu.

2 Tietoperusta

2.1 Funktionaalinen ohjelmointi

Funktionaalinen ohjelmointi on deklaratiivisen ohjelmoinnin alalaji, jossa ohjelma ilmaistaan funktioiden avulla. Funktionaalisen ohjelmoinnin piirteisiin kuuluu mm. funktioiden käyttö argumentteina toisissa funktioissa, funktioiden palauttaminen toisista funktioista ja tilan muutosten minimoiminen. Keskeisenä asiana on uusien arvojen luominen vanhoista funktioiden avulla. Eräs tyypillisesti käytettävä tekniikka on pattern-matching, jonka avulla pystytään tarkastelemaan, vastaako argumenttina annettu arvo jotakin määritetyistä kaavoista. (Watt 2004, 367-376.) Eräitä tässä opinnäytetyössä usein käytettyjä funktionaaliseen ohjelmointiin liittyviä funktioita ovat mm. map, flatMap, zip ja filter.

2.2 Scala

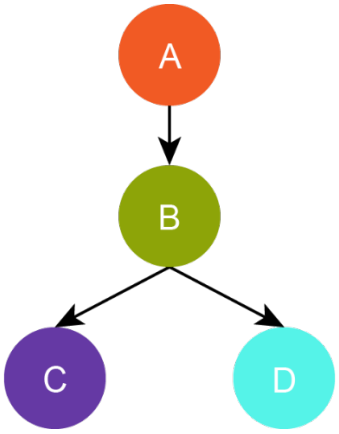
Scala on Java Virtual Machinella toimiva moniparadigmainen ohjelmointikieli, jossa yhdistyvät muun muassa funktionaalinen ohjelmointi olio-ohjelmoinnin kanssa. Scalalla kirjoitettu koodi kääntyy Java Byte Codeksi, joten se on yhteensopiva Java-luokkien kanssa. Funktionaaliseksi kielen tekee tuki funktioiden käytölle argumentteina toisissa funktioissa. (Odersky ym. 2004, 1-2.) Lisäksi henkilökohtaisena kokemuksena on, että Scala suosii funktionaaliseen ohjelmoinnin tapaisesti muuttumattomien arvojen käyttöä muuttujien sijaan mm. tarjoamalla muuttumattomia versioita Javan kokoelmista, kuten listoista.

2.3 Reaktiivinen ohjelmointi

Reaktiivinen ohjelmointi kuvataan varhaisissa julkaisuissa reaktiivisen järjestelmän ohjelmoimisena. Boussinotin (1991, 401) mukaan reaktiivinen järjestelmä koostuu interaktiivisista ohjelmista, jotka ovat jatkuvassa vuorovaikutuksessa ulkoisen lähteen kanssa, ja reagoivat siltä saatuihin aktivointeihin. Reaktiivisessa ohjelmoinnissa puolestaan määritetään näitä reaktioita. Toisin sanoen reaktiivinen ohjelmointi on tapahtumapohjaista. Cooperin (2008, 1) mukaan oikeastaan suurin osa ohjelmista on reaktiivisia. Graafinen käyttöliittymä on esimerkki tällaisesta reaktiivisesta ohjelmasta.

Bainomugishan ym. (2013, 2-3) mukaan reaktiivisessa ohjelmoinnissa on kyse perinteisen *takaisinkutsuihin* perustuvan tapahtumapohjaisen ohjelmoinnin korvaamisesta (Takaisinkutsuja käsitellään tarkemmin kappaleessa 2.4). Tässä näkemyksessä reaktiivinen ohjelmointi on paradigma, joka abstraktoi ajan hallinnan samalla tavalla kuin roskien keräys (garbage collection) muistin hallinnan tukemalla ainakin teoriassa samanaikaisuutta. Lisäksi tässä näkemyksessä määritetään kaksi keskeistä piirrettä reaktiiviselle ohjelmoin-

nille: *jatkuvaan aikaan (continuous time)* perustuvat muuttuvat arvot ja *muutosten propagointi (propagation of change)*. Jatkuvalla ajalla tarkoitetaan tässä yhteydessä sitä, että funktiolla on loputon määrä arvoja tietyllä aikavälillä. *Muutosten propagoinnilla (propagation of change)* tarkoitetaan, että muutokset arvossa vaikuttavat automaattisesti kaikkiin arvoihin, jotka ovat siitä riippuvaisia. Esimerkiksi kuvassa 1: Jos arvo B on riippuvainen A:stä ja arvot C ja D riippuvaisia B:stä, niin muutokset A:ssa päivittävät arvot C ja D.



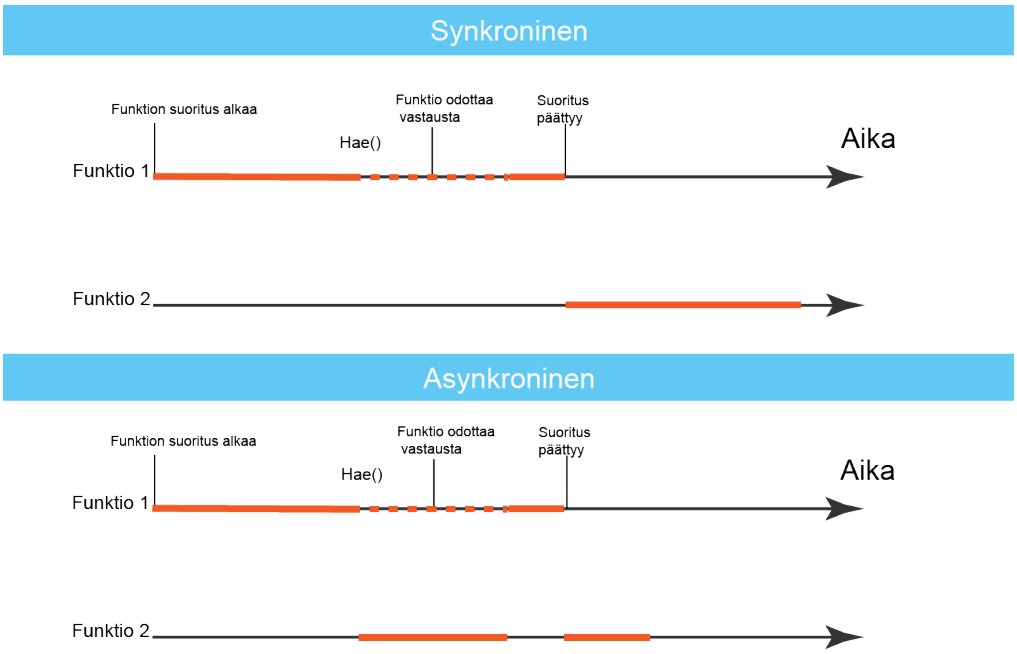
Kuva 1. Visualisointi muutosten propagoinnista

Baingomugisha ym. (2013, 12) jakavat reaktiivisen ohjelmoinnin kolmeen alalajiin: funktionaalisen reaktiivisen ohjelmoinnin (FRP:n) sisaruksiin, reaktiivisen ohjelmoinnin serkuksiin ja paradigmaan nimeltä synchronous dataflow -ohjelmointi. Näistä jälkimmäisin on varhaisin reaktiiviseen ohjelmointiin liittyvä paradigma, ja siten myös vanha, joten sen käsittely on rajattu pois tästä työstä.

Reaktiivista ohjelmointia tarkastellessa Scalalla tulee ottaa huomioon myös arkkitehtuuri/suunnittelumalli Reactive System, joka Scalaa kehittävää Lightbendiä (ent. Typesafe) edustavien Bonérin ja Klangin (2016) mukaan sekoitetaan usein reaktiivisen ohjelmoinnin kanssa. Kyseinen arkkitehtuuri on määritetty alun perin vuonna 2013 julkaistussa ja vuonna 2014 versioon 2.0 päivitetyssä reaktiivisessa manifestissa (Bonér ym. 2014).

Bonérin ja Klangin (2016) mukaan reaktiivinen ohjelmointi on alalaji asynkroniselle ohjelmoinnille ja dataflow-ohjelmoinnille – paradigmalle, jossa uuden tiedon saatavuus ajaa sovellusta eteenpäin. Asynkronisessa ohjelmoinnissa optimoidaan resurssien käyttöä pitämällä huoli siitä, että säiettä/säikeitä käytetään mahdollisimman tehokkaasti. Kuvassa 2 on esimerkki synkronisuudesta ja asynkronisuudesta. Synkronisessa odotetaan, kunnes funktion suorittaminen on loppunut säikeen etenemistä estämällä, asynkronisessa puoles-

taan säikeen etenemistä ei tarvitse estää, vaan funktioita voidaan suorittaa limittäin tai samanaikaisesti. Bonérin ja Klangin (2016) mukaan reaktiivinen ohjelmointi voi olla joko *takaisinkutsuihin* perustuvaa tai deklaratiivista. Sitä tukevia tekniikoita ovat muun muassa *futuurit/lupauks*et, *virrat* ja *dataflow-muuttujat*. Futuureja/lupauksia sekä dataflow-muuttujia käsitellään lisää kappaleessa 2.6, ja virtoja kappaleessa 2.7.



Kuva 2. Synkroniset funktiot vs. asykroniset yhdellä säikeellä

Keskeisenä erona reaktiivisen ohjelmoinnin ja Reactive System -suunnittelumallin välillä on paradigmoissa, joita ne noudattavat. Reaktiivinen ohjelmointi on tapahtumapohjaista, ja Reactive System viestipohjaista. Erona tässä on se, että viesteillä on yksi tietty vastaanottaja, ja tapahtumilla puolestaan ei ole tiettyä vastaanottajaa. Reactive System -mallin mukaiset komponentit voidaan kuitenkin toteuttaa reaktiivisella ohjelmoinnilla. Se ei ole kuitenkaan välttämättömyys. (Bonér & Klang 2016.)

2.4 Takaisinkutsut

Takaisinkutsut (callbacks) ovat imperatiiviseen, tapahtumapohjaiseen ohjelmointiin, kuuluva tekniikka (Cooper 2008, 3). Takaisinkutsuissa on kyse funktion tai olio-ohjelmoinnin tapauksessa rajapinnan metodin käyttämisestä argumenttina funktiossa, joka kutsuu sitä myöhemmin. Kuvassa 3 on yksinkertainen esimerkki takaisinkutsusta. Siinä funktio ottaa argumentiksi merkkijonon ja takaisinkutsufunktion, ja suorittaa sen jokaiselle merkkijonon kirjaimelle. Takaisinkutsufunktioksi on määritetty argumentin tulostava anonyymi funktio.

```
def forEachChar(str: String, callback: Char => Unit): Unit = {
  for (char <- str) {
    callback(char)
  }
}

var done = false

while (!done) {
  val input = scala.io.StdIn.readLine()

  input match {
    case a if a.isEmpty => done = true
    case s => forEachChar(s, x => println(x))
  }
}
```

Kuva 3. Yksinkertainen esimerkki takaisinkutsusta

Takaisinkutsuihin liittyy lukuisia ongelmia, jotka yhdistetään arkikieliseen termiin Callback Hell (Edwards 2009, 2). Eräs niistä on sivuvaikutukset: Takaisinkutsut evät palauta arvoja, vaan ne muuttavat tilaa. Lisäksi ne kääntävät ohjelman rakenteen – olio joutuu tarkkailemaan tilansa muutoksia ja päivittämään toisia siitä riippuvaisia olioita. Tämä saattaa tehdä oliota muuttavasta koodista vaikean havaita ja siten koodista vaikeasti ymmärrettävää. (Cooper 2008, 3-5.) Takaisinkutsuja käytettäessä väärin voidaan päätyä ongelmaan, jossa funktioiden sisäkkäisyys vaikeuttaa koodin luettavuutta. Kuvasta 4 voidaan nähdä liioiteltu esimerkki tästä ongelmasta.

```
def func1(callback: String => Unit) = callback("this")
def func2(str: String, callback: String => Unit) = callback(str + "is")
def func3(str: String, callback: String => Unit) = callback(str + "callback")
def func4(str: String, callback: String => Unit) = callback(str + "hell")
def func5(str: String, callback: String => Unit) = callback(str + "!")

def callbackHell() = {
  func1(a => {
    func2(a + " ", b => {
      func3(b + " ", c => {
        func4(c + " ", d => {
          func5(d, e => println(e.toUpperCase))
        })
      })
    })
  })
}
callbackHell()
```

Kuva 4. Esimerkki Callback Hell -ongelmasta

2.5 Tarkkailijamalli ja Reactor-malli

Tarkkailijamalli (observer pattern) on olio-ohjelmointiin kuuluva suunnittelumalli, joka pohjautuu takaisinkutsuihin. Tarkkailijamalli mahdollistaa olion tilan muuttumisen ilmoittamisen automaattisesti tarkkailijoilta (observers) tarkkailtaville kohteille (subjects). Näistä käytetään myös termejä tilaaja (subscriber) ja tarjoaja (provider). Tarkkailijamallin tarkoituksena on parantaa olioiden uudelleenkäytettävyyttä vähentämällä olioiden riippuvuutta toisistaan (loose coupling). (Gamma ym. 1995, 293-294.) Maierin ja Oderskyn (2012, 1) mukaan tarkkailijamalli on yksi hallitsevista tavoista hallita ohjelman tilan muutoksia mm. käyttöliittymissä.

Kuvassa 5 on tarkkailijamallin mukainen rajapinta. Siinä on määriteltynä rajapinnat tarkkailijalle ja tarkkailtavalle kohteelle. Tarkkailtava kohteen rajapinta sisältää mm. tarkkailijoiden lisäämisen ja poistamisen, sekä tarkkailijoiden ja tilan päivittämisen. Takaisinkutsuihin pohjautuminen tulee esille tarkkailijoille muutoksia ilmoitettaessa update-takaisinkutsumetodin avulla.

```
trait Observer[T] {
  def update(t: T): Unit
}

trait Subject[T] {
  var observers: mutable.MutableList[Observer[T]] = new mutable.MutableList[Observer[T]]

  var state: T

  def addObserver(observer: Observer[T]): Unit = {
    observers += observer
  }

  def removeObserver(observer: Observer[T]): Unit = {
    observers = observers.collect { case o if o != observer => o }
  }

  def notifyObservers(): Unit = {
    observers.foreach(_.update(state))
  }

  def setState(state: T): Unit = {
    this.state = state

    notifyObservers()
  }

  def getState: T = {
    this.state
  }
}
```

Kuva 5. Tarkkailijamallin mukaiset rajapinnat

Kuvassa 6 esitellään esimerkki kuvan 5 rajapintojen toteuttamisesta ja käyttämisestä. Kyseinen esimerkki tulostaa käyttäjän antaman syötteen. Siinä toteutetaan tarkkailtava

kohde, jonka tilaa muutetaan silmukassa käyttäjäsyötteen mukaan. Tarkkailija päivittyy tilan muutoksen seurauksena ja tulostaa syötteen.

```
object Main {
  def main(args: Array[String]): Unit = {
    val subject = new Subject[String] { var state = "" }

    val observer = new Observer[String] {
      override def update(str: String) = {
        println("Received input: " + str)
      }
    }

    subject.addObserver(observer)

    var done = false
    while (!done) {
      print("Input a line: ")
      val input = scala.io.StdIn.readLine()
      subject.setState(input)

      input match {
        case s if s.isEmpty => done = true
        case _ =>
      }
    }
  }
}
```

Kuva 6. Esimerkki tarkkailijamallin käytöstä

Tarkkailijamalli on helppo toteuttaa, mutta se perustuu takaisinkutsuihin, joten sitä vaivaavat osittain samat ongelmat. Ingo Maierin ja Martin Oderskyn (2012, 1-13) mukaan näitä ovat muun muassa:

- Sivuvaikutukset: mm. tilan muuttuminen aiheuttaa sivuvaikutuksia.
- Enkapsulointi: tarkkailtavan kohteen tilan muuttuja on näkyvissä kaikille tarkkailijoille.
- Kompositio: Uusia ominaisuuksia on vaikea koostaa aiemmin määritettyjä tarkkailijoita käyttäen.
- Resurssien hallinta: JVM:n roskienkerääjä (garbage collector) ei osaa hallita tarkkailijan resursseja oikein, mikä voi aiheuttaa muistivuodon (memory leak). Tarkkailijan elämänkaarta (life-cycle) täytyy hallita siis eksplisiittisesti.

Reactor-malli (Reactor pattern) on tarkkailijamalliin liittyvä suunnittelumalli. Erona on kuitenkin se, että tarkkailijamallissa tarkkailijoita informoidaan yksittäisen tarkkailtavan kohteen muuttuessa. Reactor-mallissa puolestaan tarkkaillaan tapahtumia keskitetysti useasta tapahtumalähteestä, ja toimitetaan ne kyseisen tapahtuman käsittelijälle. (Schmidt 1995, 10.)

2.6 Futuurit ja dataflow-muuttujat

Futuurit (futures) mahdollistavat usean laskutoimituksen ajamisen samanaikaisesti samalla välttäen säikeen etenemisen estämistä. Lisäksi ne mahdollistavat asynkroniset arvon muunnokset ennen kuin se on saatavilla. Futuureiden avulla voidaan toteuttaa siis sekä samanaikaista, että asynkronista ohjelmointia. Scalassa futuurit hyödyntävät yleensä takaisinkutsuja, mutta ne mahdollistavat yksinkertaisemman koodin komposition, eli futuurien koostamisen ansiosta. Scalassa futuureita voidaan koostaa muun muassa funktioiden flatMap, foreach ja filter avulla. (Bonér & Klang 2016; Haller ym. 2015.) Kuvassa 7 on käytetty for -komprehensiota futuurien koostamiseen. Se on syntaktista sokeria, ja saman asian saa aikaiseksi flatMap -funktioiden avulla. Kuviosta nähdään myös se, että futuurin arvon käyttäminen tapahtuu takaisinkutsuihin pohjautuvan onComplete-metodin sisällä.

```
case class Pizza(crust: Crust, toppings: List[Topping])

def bakePizza(): Future[Pizza] = {
  val crust: Future[Crust] = prepareCrust()
  val toppings: Future[List[Topping]] = fetchToppings()

  for {
    c <- crust
    t <- toppings
    p = Pizza(c, t)
  } yield p
}

bakePizza().onComplete {
  case Success(p: Pizza) => deliverPizza(p)
  case Failure(f) => throw f
}
```

Kuva 7. Futuurien kompositio

Bainomugisha ym. (2013) eivät ota kantaa futuureihin, mutta voitaisiin ehkä ajatella, että niillä on joitakin reaktiivisen ohjelmoinnin piirteitä. Asynkronisuuden ansiosta ne abstraktoivat ajan hallintaa – ohjelmoija ei tiedä milloin ne suoritetaan, vain järjestyksen. Lisäksi ne saatetaan ajaa automaattisesti samanaikaisesti (Haller ym. 2015). Myös muutosten propagoinnin toteutuminen voidaan nähdä mahdollisena, kun futuureja komposoidaan funktionaalisesti.

Dataflow-muuttujat (dataflow-variables) ovat futuurien kaltaisia asynkronisen ohjelmoinnin abstraktioita, erona kuitenkin se, että futuureihin pystyy kirjoittamaan vain kerran, ja dataflow-muuttujiin voidaan kirjoittaa useita kertoja. Ne ovat siis muuttujia, jotka reagoivat

automaattisesti muutoksiin. Lisäksi ne poistavat ohjelmoijan määrittämät staattiset riippuvuudet dynaamisilla riippuvuuksilla, jotka määräytyvät ohjelmassa määritellyn tiedon mukaan. (Van-Roy & Haridi 2004, 335-336.) Tämä voidaan nähdä tapana, millä dataflow-muuttujat toteuttavat muutosten propagoinnin.

2.7 Virrat

Bonérin ja Klangin (2016.) mukaan virrat (streams) tarkoittavat rajattoman tietomäärän prosessointia. Scala 2.12 dokumentaatiossa virrat kuvataan Stream-luokassa. Stream-luokka implementoi sekvenssin, jonka arvot lasketaan vasta, kun niitä tarvitaan. Se käyttää siis laiskaa evaluointia. Toisin kuin listoilla (esim. List tai Seq) sillä on mahdollista olla loputon määrä arvoja. Jos arvot säilytetään, eli toteutetaan memoisaatio (memoization), niin muisti voi kuitenkin loppua kesken. (EPFL 2016.) Kuvassa 9 on esimerkki Scalan Stream-luokan käytöstä fibonaccin lukujonon 1, 1, 2, 3, 5, 8... ensimmäisten 10 numeron laskemiseen.

```
def fibonacci: Stream[BigInt] = {
  Stream.continually((BigInt(0), BigInt(0)))
    .scan((BigInt(0), BigInt(1)))((a, b) => (a._2, a._1 + a._2))
    .map(_._2)
}

fibonacci.take(10).foreach(println(_))
```

Kuva 9. Fibonaccin lukujonon laskemista Stream-luokan avulla

2.8 Reaktiivisen ohjelmoinnin evaluaatiomallit

Evaluaatiomalli kertoo, kuinka muutokset propagoidaan riippuvuuksien välillä. Ohjelmoijan näkökulmasta tämä tapahtuu automaattisesti. Tällä on kuitenkin vaikutusta siihen, miten ohjelma toimii. Tieto tapahtumasta siirtyy tapahtumalähteeltä (event source), eli tuottajalta/julkaisijalta (producer), kuluttajille/käsittelijöille (consumers). On kuitenkin kaksi erilaista tapaa, miten tieto muutoksesta saadaan. *Pull-menetelmässä* kuluttajat ”vetävät” tietoa tuottajalta. Pull-menetelmän varjopuolena on viive tapahtuman esiintymässä ja sen reaktiossa. *Push-menetelmä* toimii päinvastaisesti kuin pull-menetelmä. Siinä tuottaja, eli tapahtumalähde, ”työntää” tiedon kuluttajille. (Bainomugisha ym. 2013, 5-6.)

2.9 Funktionaalinen reaktiivinen ohjelmointi

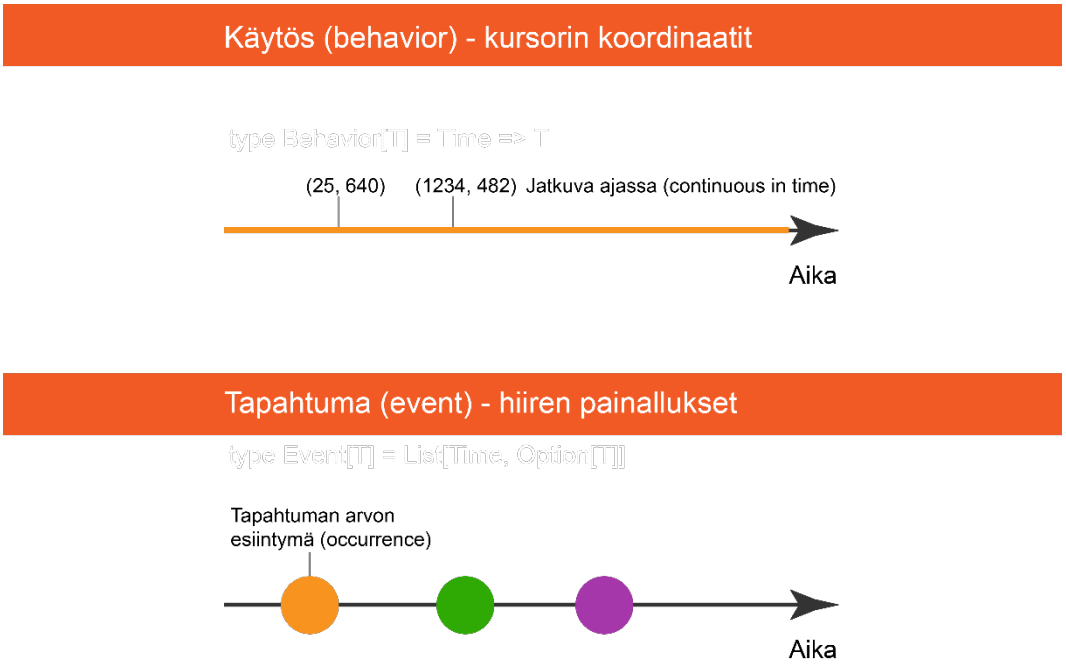
Funktionaalinen reaktiivinen ohjelmointi (FRP) on ohjelmointiparadigma reaktiivisten järjestelmien ohjelmoimiseen deklaratiivisesti. FRP toteutetaan usein toisen ohjelmointikielen sisällä, eli *DSL:nä* (domain specific language). Paradigmasta on useita versioita, ja ne on esitelty tyypillisesti puhtaasti funktionaalisella Haskell-ohjelmointikielellä. (Wan & Hudak 2000, 1-2; Nilsson ym. 2002, 1.) FRP perustuu Conal Elliottin vuonna 1997 julkaisemaan FRAN:iin, joka on ohjelmistokehys Haskellille multimedia-animaatioiden tekemiseen. FRP:tä on käytetty mm. käyttöliittymissä, robotiikassa ja web-ohjelmoinnissa. (Hudak ym. 2003, 2; Nilsson ym. 2002, 1.)

Blackheath ja Jones (2016) luettelevat FRP:tä käsittelevässä kirjassaan kappaleessa 1 kyseiselle paradigmalle erilaisia piirteitä, ja miten se voidaan nähdä nykyaikana:

- Sillä voidaan korvata laajasti käytössä oleva tarkkailijamalli (observer-pattern).
- Se on modulaarinen tapa ohjelmoida tapahtumapohjaista logiikkaa.
- Ohjelmat voidaan esittää reaktioina syötteisiin tai tiedon virtana (flow of data).
- Se poistaa implisiittisen tilan (state) funktionaalisuuden avulla.
- Vastaavaan ratkaisuun päädytään väistämättä, kun yritetään korjata tarkkailijamallin ongelmia.
- Se voidaan toteuttaa DSL:nä toisessa kielessä.

FRP:n luojan Conal Elliottin (13.6.2015; 2015) mukaan FRP on yhä etenevissä määrin ymmärretty väärin, ja sitä käytetään kuvaamaan virheellisesti mm. Elm-ohjelmointikieltä, Bacon -JavaScript-kirjastoa ja Scalallekin käännettyä Reactive Extensions -kirjastoa. Elliott määrittää FRP:n keskeisimmiksi piirteiksi jatkuvan ajan (continuous time) ja yksinkertaisen matemaattisen määritelmän.

FRP:llä on selvä matemaattinen määritelmä, ja siinä on yksi tietotyyppi – käytös (behavior). Käytös määritetään seuraavasti: λ::Behavior a -> (T -> a), jossa T on aikaa kuvaava reaaliluku. Eli käytös on ajasta riippuvainen funktio, joka ottaa reaaliluvun T ja palauttaa arvon a. Esimerkki käytöksestä voisi olla kursorin sijainti ruudulla. Hiirtä liikuteltaessa kursorin koordinaatit ovat jokaisena tarkasteltavana hetkenä eri. Tapahtuma on käytöksen alatyyppi. Se on diskreetti, eli sillä on arvo vain tiettyinä hetkinä. (Elliott 13.6.2015; Elliott 2015.) Tapahtumaa voisi kuvata esimerkiksi painikkeen painallus käyttöliittymässä tai HTTP-kutsun vastaanottaminen. Kuvassa 8 on kuvattuna esimerkit käytösten jatkuvuudesta, ja tapahtumien diskreettiydestä. Kuvassa on esitettynä hiiren kursorin sijaintia kuvaava käytös ja hiiren painalluksen aikaan saama tapahtuma.



Kuva 8. Käytökset ja tapahtumat

Käytösten ja tapahtumien lisäksi FRP tarjoaa määrittelyt funktioille, joiden avulla pystytään "nostamaan" tavallisia arvoja käytöksiksi, yhdistämään käytöksiä ja vaihtamaan käytöstä tapahtuman sattuessa. Nykyaikaisen funktionaalisen ohjelmoinnin terminologian mukaan nämä ovat funktoreita, applikatiivisia funktoreita (applicative functors) ja monoideja. (Elliott 13.6.2015; Elliott 2015; Elliott 2009, 27-29.)

FRP:stä on kehitetty erilaisia versioita, joita käytetään hiukan eri tavalla. Ensimmäisen version ns. klassisen FRP:n jälkeen ehdotetu versiot ovat yhdistäneet käytökset ja tapahtumat yhteen tietotyyppiin, joka kulkee usein nimellä Signal. Erilaisten abstraktioiden kehittämistä on ajanut erilaiset ongelmat versioissa. Esimerkiksi varhaisissa pull-menetelmään perustuvissa versioissa esiintyy time-leak ja space-leak -ongelmia, jotka ilmenevät muistivuotoina, kun ohjelmaa on ajettu tarpeeksi kauan. Klassisen FRP:n jälkeisiä toteutuksia ovat mm. Yampa ja Push-Pull FRP. Yampassa ongelmat on ratkaistu rajoittamalla ilmaisumahdollisuuksia ja Push-Pull FRP:ssä yhdistämällä push- ja pull-menetelmiä. (Elliot 2009, 1-2; Hudak ym. 2003, 4.)

2.10 Luokittelematon/muu reaktiivinen ohjelmointi

Bonérin ja Klangin (2016) mukaan funktionaalista reaktiivista ohjelmointia ei pidä sekoittaa reaktiiviseen ohjelmointiin. Heidän mukaan suurin osa FRP ratkaisuista ovatkin todellisuudessa reaktiiviseen ohjelmointiin kuuluvia. On epäselvää, että tarkoitetaanko tällä sitä, että FRP ei olisi reaktiivisen ohjelmoinnin alalaji vai, että se olisi käytännön tasolla niin kaukana muusta reaktiivisesta ohjelmoinnista, että yhteys jää vain käytettävän termin samankaltaisuuteen.

Bainomugisha ym. (2013, 10) luokittelevat muut kuin FRP:hen tai synchronous dataflow -ohjelmointiparadigman toteuttavat kielet reaktiivisen ohjelmoinnin serkuksiksi. Termi on mahdollisesti harhaanjohtava, joten tässä opinnäytetyössä käytetään termiä luokittelematon/muu reaktiivinen ohjelmointi. Kyseinen alaluokka sisältää tekniikat, joita ei ole pystytty luokittelemaan tarkasti kirjallisuudessa. Tässä opinnäytetyössä siihen liitetään myös Bonérin ja Klangin (2016) näkemys reaktiivisesta ohjelmoinnista.

2.11 Reactive Streams -standardi

Reactive Streams on ehdotus standardiksi JVM:lle toteuttaa virtoihin perustuva ratkaisu tiedon välitykselle reaktiivisen ohjelmoinnin kirjastojen välillä asynkronisesti. Se laajentaa reaktiivisessa manifestissa määritellyn asynkronisen komponenttien viestien välityksen koskemaan ohjelman sisäistä viestien välitystä. (Bonér & Klang 2016.) Standardiehdotus on toistaiseksi toteutumassa osaksi Java 9 -standardia Flow-API:ssa (Oracle 2016). Reactive Streams -standardia on ollut toteuttamassa muun muassa Lightbendin, Netflixin, Red Hat:in ja Twitterin suunnittelijat. (Reactive Streams Special Interest Group 2016.) Reactive Streams sivustolla mainitaan standardin toteuttaviksi teknologioiksi seuraavia toteutuksia: Akka Streams, MongoDB, Ratpack, Reactive Rabbit, Reactor, RxJava, Slick ja Vert.x 3.0. (Reactive Streams Special Interest Group 2015.)

Reactive Streams on tarkoitettu alhaisen tason ohjelmointiin, ja sen sijaan, että sitä käytettäisiin suoraan, se on ensisijaisesti tarkoitettu kirjastojen toteuttamiseen (Reactive Streams Special Interest Group 2016). Sen päätehtävät on kuitenkin hyvä ymmärtää siihen pohjautuvia kirjastoja käytettäessä. Reaktiivisen virran keskeisenä asiana on ottaa huomioon tietovirtojen siirrossa, että virran lähde voi siirtää tietoa nopeammin kuin sen vastaanottaja pystyy käsittelemään. Tätä nimitetään termillä backpressure.

Reactive Streams-standardin mukainen backpressure auttaa ylläpitämään sovelluksen palautumista (resilience), ennalta-arvattavuutta ja suorituskykyä. Backpressure mahdollistaa asynkronisten virtojen virheiden viestittämisen, ja niistä palautumisen. (Kapadia 2016.) Esimerkiksi virran käsittelijällä voi viestittää lähteelle, mikäli sen muisti on loppumassa. Tällöin voidaan hallitusti joko viestittää virheestä, jolloin tiedon siirtäminen loppuu tai jos kyse ei ole tärkeästä tiedosta, niin se voidaan jättää käsittelemättä.

Reactive Streams-standardin API koostuu neljästä komponentista: Publisher, Subscriber, Subscription ja Processor. Publisher tarjoaa potentiaalisesti loputtoman määrän elementtejä, jotka julkaistaan Subscriber:in tarpeen mukaan. Subscriber viestittää Publisherille Subscriptionin request-metodin avulla, kuinka monta elementtiä se kykenee käsittelemään. (Reactive Streams Special Interest Group 2016.) Rajapinnoista voidaan havaita, että ne muistuttavat hiukan tarkkailija- tai Reactor-mallia publisher-subscriber osalta.

Reactive Stream backpressure toteutetaan dynaamisella pull-push-mallilla (Reactive Streams Special Interest Group 2016). Ratkaisusta on mahdollisesti pääteltävissä, että evaluaatiomalli on samanlainen: Käsittelijä pyytää tietyn määrän elementtejä lähteeltä, joka työntää elementtejä, kunnes kyseinen määrä täyttyy.

3 Empiirinen osa

3.1 Tutkimusongelmat

Tutkimuksessa perehdyttiin reaktiivisen ohjelmoinnin kirjastoihin Scalalla. Tätä varten selvitettiin mitä kirjastoja on tarjolla, ja mihin teoriassa esitettyihin tekniikoihin ne pohjautuvat. Näiden lisäksi tarkoituksena oli havaita mitä reaktiivisen ohjelmoinnin piirteitä ne toteuttavat, ja mihin alalajiin ne kuuluvat. Lisäksi tutkimuksessa perehdyttiin syvemmin näiden joukosta valittujen kirjastojen käyttöön, ja pyrittiin havaitsemaan, millainen vaikutus niiden käytöllä on koodirivien määrään, ja siten koodin luettavuuteen yksinkertaisessa sovelluksessa. Tämän kaiken tarkoituksena oli arvioida reaktiivisen ohjelmoinnin käyttökelpoisuutta työelämää varten.

3.2 Menetelmät

Tutkimus toteutettiin vertailulla. Ensin tutustuttiin yleiskäyttöisiin reaktiivisen ohjelmoinnin kirjastoihin. Joukosta valittiin yksi, jonka avulla toteutettiin yksinkertainen asynkroninen REST-pohjainen microservice, jota verrattiin natiivin Scalan futuureilla toteutettuun versioon. Tutkimuksessa verrattiin koodirivien lukumäärää toteutusten kesken. Reaktiivisen ohjelmoinnin tarkoituksena on helpottaa tapahtumapohjaisen sovellusten tekemistä mm. helpottamalla koodin luettavuutta. Tähän vaikuttaa osittain koodirivien lukumäärä. Lisäksi koodirivien lukumäärä on sidoksissa virheiden määrään. Koodirivien lukumäärän lisäksi kirjaston käytön helppoutta ja koodin luettavuutta arvioitiin subjektiivisesti.

3.3 Reaktiivisen ohjelmoinnin kirjastot Scalalle

Reaktiivista ohjelmoimista tarkasteltaessa Scalalla ja JVM:llä pitää ottaa huomioon se, että kaikissa reaktiiviset ratkaisut eivät ole varsinaisesti tarkoitettu reaktiiviseen ohjelmoimiseen, vaan osa on reaktiivisen manifestin mukaisia reaktiivisia järjestelmiä varten. Esimerkiksi Akka Actor on tarkoitettu viestipohjaiseen ohjelmoimiseen tapahtumapohjaisen sijaan. Sen tehtävänä on välittää viestejä reaktiivisen järjestelmän eri komponenttien välillä. Sen sijaan Akka Streams on virtoihin pohjautuva kirjasto, joka toteuttaa Reactive Streams -standardin.

Tässä luvussa on esiteltynä tutkimusta varten löydettyjä kaupallisesti merkittäviä reaktiivisen ohjelmoinnin kirjastoja JVM:llä. Tässä yhteydessä sillä tarkoitetaan, että kirjaston takana on merkittävä organisaatio tai kirjastoa käyttää tunnetusti suuri(a) yritys/yrityksiä. Poikkeuksena esitellään Sodium, joka on eräs varteenotettavimmista funktionaalisen reaktiivisen ohjelmoinnin kirjastoista JVM:llä. Lisäksi esitellyissä kirjastoissa kriteerinä oli yleiskäyttöisyys. Tällä pyrittiin rajaamaan tutkimuksen ulkopuolelle esimerkiksi tietokantakirjastot, kuten johdannossa mainittu Slick.

3.3.1 Reactive Extensions (RxScala/RxJava)

Reactive Extensions on alun perin .NET Frameworkille toteutettu kirjasto. Kyseinen kirjasto on laajasti käytössä. Sen käyttäjiin lukeutuvat mm. Microsoft, Netflix ja Github. JVM-kielten, kuten Java, Scala, Groovy ja Clojure, lisäksi kirjasto on saatavissa useille eri kielille, kuten C#:lle C++:lle, JavaScriptille ja Pythonille. Kirjasto on inspiroitunut tarkkailijamallista, ja yhdistää siihen funktionaalisen ohjelmoinnin ideoita. (ReactiveX 2017a.) Dokumentaatiosta selviää kuitenkin tarkennus, että tarkkailijamallin sijaan se pohjautuu Reactor-malliin (ReactiveX 2017b).

Huomioitavaa on, että RxJava toteuttaa Reactive Streams -standardin versiossa 2.0, mutta uusin versio RxScala 0.26.5 pohjautuu vanhempaan RxJava versioon 1.2.4, joten se puuttuu kyseisestä versiosta. Tämän vuoksi esitellään RxJava-versio, sillä oletettavasti RxScala tulee toteuttamaan kyseisen standardin lähitulevaisuudessa.

RxJava toteuttaa Bainomugisha ym. (2013, 3) mainitsemista reaktiivisen ohjelmoinnin kriteereistä automaattisen muutosten propagoinnin, mutta ei jatkuvaan aikaan perustuvaa abstraktiota. Julkaisussa käsitellään .NET versiota, ja siinä todetaan propagoinnin tapahtuvan push-menetelmällä. RxJavan uusin versio kuitenkin toteuttaa Reactive Streams –rajapinnan, joten oletettavaa on, että siinä käytetään push-pull -menetelmää. Kirjasto toteuttaa myös Bonerin ja Klangin (2016) mainitseman kriteerin asynkronisuudesta. Nämä yhdessä tekevät kirjastosta luokittelemattomaan/muuhun reaktiiviseen ohjelmointiin kuuluvan.

RxJavassa yksi pääabstraktioista on Observable, joka vastaa (tapahtuma)virtojen tarkkailusta ja muuttamisesta. Observable-luokka tarjoaa metodeita kompositioon ja muita funktionaalisesta ohjelmoinnista inspiroituneita funktioita, kuten flatMap, map ja zip. Lisäksi luokka tarjoaa subscribe-metodin, jonka avulla elementtejä voidaan käyttää niiden saapuessa käytettäväksi. Kuvassa 10 on esiteltynä kirjaston syntaksista fibonaccin lukujonon ensimmäisen 10 numeron laskemisessa kirjastoa käyttäen.

```
def fibonacciRx: Observable[BigInt] = {
    Observable.just((BigInt(0), BigInt(0))).repeat()
      .scan((BigInt(0), BigInt(1)), (a: (BigInt, BigInt), _) => (a._2, a._1 + a._2))
      .map(_._2)
}

fibonacciRx.take(10).subscribe(println(_))
```

Kuva 10. Yksinkertainen esimerkki RxJavan syntaksista

RxJavassa/RxScalassa on laaja dokumentaatio ja sitä ei ole sidottu tiettyyn kirjastojen kokoelmaan tai ohjelmistokehykseen, mitkä tekevät siitä varman valinnan erityyppisiin projekteihin. Lisäksi eri kielille tehdyt toteutukset muistuttavat käytöltään toisiaan, mikä voi olla ratkaiseva tekijä projektien teknologioita valittaessa.

3.3.2 Akka Streams

Akka Streams on Akka-tuotteeseen/tuoteperheeseen kuuluva virtoihin perustuva reaktiivinen kirjasto Scalalle ja Javalle. Akka tunnetaan hajautettuja järjestelmiä varten toteutetusta Akka Actor -kirjastosta. Akka on osa Lightbendin Reactive Platform – reaktiivisen manifestin mukaisen reaktiivisen järjestelmän toteuttamiseen tarkoitettavaa tuoteperhettä. Akka Streams tuo tähän lisänä reaktiivisen ohjelmoinnin. Kirjaston versio 1.0 julkaistiin vuonna 2015 (Malawski 2015). Nimenomaista kirjastoa käyttävistä yrityksistä ei ole tietoa, mutta Akka-tuotetta käyttävät muun muassa Amazon.com, ebay, BBC ja The Guardian. Lisäksi kirjaston päälle on tehty Akka HTTP. (Lightbend inc 2016.)

Akka Streams toteuttaa Reactive Streams -rajapinnan. Sen omat abstraktiot eivät kuitenkaan periydy siitä, vaan se tarjoaa metodeita, joilla saadaan muutettua ne Reactive Streams-rajapintaa vastaaviin abstraktioihin. Publisher-abstraktiota vastaa Source, Subscriber-abstraktiota Sink ja Processor-abstraktiota Flow. Kuvassa 11 on yksinkertainen esimerkki kirjaston syntaksista.

```
def fibonacciAkka: Source[BigInt, NotUsed] = {
  Source.repeat((BigInt(0), BigInt(0)))
    .scan((BigInt(0), BigInt(1)))((a, b) => (a._2, a._1 + a._2))
    .map(_._2)
}

fibonacciAkka.take(10).to(Sink.foreach(println(_))).run()
```

Kuva 11. Yksinkertainen esimerkki Akka Streams -kirjaston syntaksista

Akka Streams on ensisijaisesti suunnattu pohjaksi kirjastoja varten, joten se on tietoisesti suunniteltu olemaan minimaalinen ja johdonmukainen, mutta ei välttämättä helppokäyttöinen. (Lightbend inc 2017.) Akka Streams poikkeaakin Reactive Extensionin käytöstä erityisesti monimutkaisia ratkaisuja toteutettaessa. Perusabstraktioiden lisäksi kirjasto tarjoaa monimutkaisemman DSL:n, jossa virroista muodostetaan graafeja, joissa käsitellään virtojen sisään- ja ulostuloja muun muassa yhdistelemällä ja haarauttamalla niitä.

Reaktiivisen ohjelmoinnin piirteiltään Akka Streams muistuttaa paljon Reactive Extensionia. Se on asynkroninen ja keskittyy muutosten propagointiin, mikä tulee erityisen selväksi graafeihin perustuvaa DSL:ää käytettäessä.

3.3.3 Apache Spark Streaming

Reaktiivisista ratkaisuista Scalalle puhuttaessa pitää ottaa huomioon suurten tietomäärien prosessointiin, eli Big Dataa varten, kehitetty Apache Spark. Siihen sisältyy kirjasto Spark Streaming, joka mahdollistaa Big Datan käsittelyn virtojen avulla. Kirjasto ei toteuta Reactive Streams -standardin mukaista rajapintaa, mutta se toteuttaa standardissa esitellyn backpressuren. Apache Spark toteuttaa reaktiivisen ohjelmoinnin kriteerien lisäksi reaktiivisen manifestin mukaisen Reactive Systems -arkkitehtuurin/mallin kriteerit, ja se on osa Lighbendin Reactive Platform -tuoteperhettä.

Apache Spark soveltuu erityisen hyvin suurten tietomäärien käsittelyyn, ja sen valinta voi olla ylilyönti, mikäli tätä ominaisuutta ei tarvita. Lisäksi kirjaston yhdistäminen toisiin reaktiivisen ohjelmoinnin kirjastoihin voi olla työlästä Reactive Streams -standardin tuen puutteen vuoksi.

3.3.4 Vert.x Rx ja Vert.x Sync

Vert.x on Eclipse Foundationin ylläpitämä tapahtumapohjainen ja polyglotti, eli monella ohjelmointikielellä käytettävä, kokoelma kirjastoja. Vaikka Vert.x toimii JVM:llä, voi sen kanssa käyttää muitakin kieliä, kuten Javascriptiä, Rubya ja Ceylonia. Siitä ei ole kuitenkaan toistaiseksi Scalalle tarkoitettua versiota. Kokoelma tarjoaa kaksi erilaista lähestymistapaa reaktiiviseen ohjelmointiin: RxJavaan pohjautuva ja Reactive Streams -standardin toteuttava Vert.x Rx sekä Quasar-kirjastoon pohjautuva, dataflow-muuttujilla tai virroilla käytettävä, Vert.x Sync.

Vert.x Sync poikkeaa merkittävästi muista esitellyistä kirjastoista funktionaalisen ohjelmoinnin sijaan imperatiivista ja olio-ohjelmointia muistuttavan rajapinnan ansiosta. Esimerkiksi virtoja yhdistettäessä luodaan uusi Mix-luokan olio flatMap- tai zip-funktioiden käyttämisen sijaan. Lisäksi se on ainoa, joka tarjoaa kaksi erilaista lähestymistapaa reaktiiviseen ohjelmointiin. Virrat soveltuvat käytettäväksi, kun käsitellään tuntematonta tai suurta määrää tapahtumia ja dataflow-muuttujat, kun käsitellään yksittäistä tapahtumaa. Kirjaston yleiskäyttöisyyttä vähentää kuitenkin Reactive Streams -tuen puuttuminen, mikä tekee kirjastosta parhaiten soveltuvan käytettäväksi lähinnä muiden Vert.x -kirjastojen kanssa.

3.3.5 Reactor

Spring-ohjelmistokehystä kehittävän Pivotal:in Reactive Streams -standardin toteuttaava virtoihin perustuva kirjasto. Kirjasto on inspiroitunut Java/ScalaRx -kirjastojen tavoin Reactor-mallista, mistä se on saanut nimensäkin. Kirjaston on tarkoitus tulla osaksi Spring-ohjelmistokehystä mm. Big Datan käsittelyssä osana Spring XD -projektia. (Brisbin 2013.) Samoin kuin Vert.x kohdalla kirjastosta ei ole erityisesti Scalalle kirjoitettua versiota. Kirjasto pohjautuu RxJavaan, joten siinä on samantapainen, mutta hiukan poikkeava syntaksi. Vaikka kirjasto on yleiskäyttöinen, se on suunnattu käytettäväksi Spring-ohjelmistokehyksen kanssa.

3.3.6 Sodium

Sodium perustuu Ingo Maierin ja Scalan kehittäjänä tunnetun Martin Oderskyn funktionaaliseen reaktiiviseen Scala.React -kirjastoon, Haskellille tehtyyn FRP-abstraktio Yampaan ja JavaScript ohjelmistokehys Flapjaxiin. Kirjasto on saatavilla Javan ja Scalana lisäksi muille ohjelmointikielille kuten C++:lle, C#:lle, Kotlinille ja TypeScriptille. Sodium on esitellyistä kirjastoista Apache Sparkin ja Vert.x Sync:n lisäksi ainoa, joka ei toteuta Reactive Streams -standardin mukaista rajapintaa. Kirjastoa käyttävistä organisaatioista ei ole tietoa.

Kirjastossa on klassisen FRP:n tavoin eroteltu käytös ja tapahtuma. Behavior-abstraktiota vastaa Cell ja Event-abstraktiota Stream. Lisäksi kyseinen kirjasto tarjoaa matemaattiset semantiikat funktioille, mikä on yksi FRP:n kriteereistä. Kirjasto on asynkroninen ja lisäksi tapahtumia voidaan käsitellä samanaikaisesti transaktioiden ansiosta. Jokainen tapahtuma aloittaa uuden transaktion, joka estää lukkojen avulla saman arvon päivittämisen samanaikaisesti usean laskutoimituksen toimesta.

Sodiumin kanssa käytettäviä toisia kirjastoja ei löytynyt, ja lisäksi dokumentaatio on niukkaa, mikä saattaa tehdä siitä epävarman valinnan useisiin projekteihin. Lisäksi kirjaston takana ei ole suurta organisaatiota, mikä tekee sen tuesta tulevaisuudessa epävarmaa. Kirjasto soveltuukin mahdollisesti parhaiten pieniin sovelluksiin, mutta ei välttämättä käytettäväksi monimutkaisiin kaupallisiiin sovelluksiin, joissa täytyy ottaa huomioon myös muut kehittäjät.

3.3.7 Yhteenveto kirjastojen ominaisuuksista

Taulukossa 1 on esitelty yhteenveto siitä, mihin tekniikoihin kirjastot pohjautuvat, toteuttavatko ne Reactive Streams-standardin ja tapahtuuko abstraktioiden yhdistäminen funktionaalisesti.

Taulukko 1. Yhteenveto kirjastojen ominaisuuksista

Kirjasto	**Pohjatuu**	**Toteuttaa Reactive Streams -standardin**	**Funktionaalinen kompositio**
Akka Streams	Virrat	Kyllä	Kyllä
Apache Spark Streaming	Virrat	Ei	Kyllä
Reactor	Reactor-malli ja virrat	Kyllä	Kyllä
RxJava 2	Reactor-malli ja virrat	Kyllä	Kyllä
RxScala (0.26.5)	Reactor-malli ja virrat	Ei	Kyllä
Sodium	FRP	Ei	Kyllä
Vert.x Rx	Reactor-malli ja virrat	Kyllä	Kyllä
Vert.x Sync	Dataflow-muuttujat / virrat	Ei	Ei

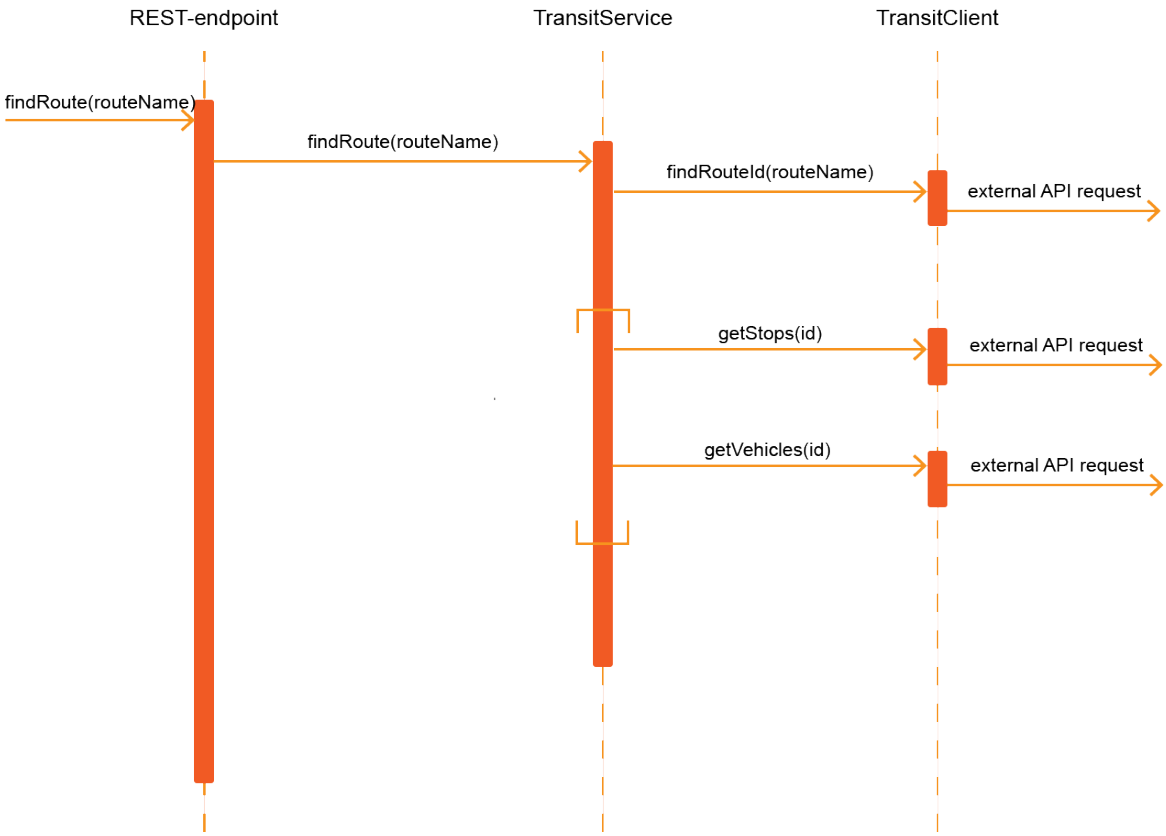
3.4 Esimerkkisovellus

Sovellus on yksinkertainen asynkronisesti toteutettu yksinkertainen REST-pohjainen microservice, joka yhdistää tietoa HSL:n ja Liikenneviraston avoimesta Digitransit rajapinnasta, ja tarjoaa sen palvelun käyttäjälle JSON-muotoisena.

Sovelluksen toteutuksessa käytettiin Scala-versiota 2.12 ja SBT-versiota 0.13.8 kääntämiseen, ajamiseen ja riippuvuuksien noutamiseen. Lisäksi sovelluksessa käytettiin Akka HTTP –kirjastoa, jota pystyy käyttämään asynkronisesti sekä futuurien avulla, että Akka Streams-virroilla. Akka Streamsin päälle tehtynä kirjastona siihen voidaan yhdistää toisia Reactive Streams -standardin toteuttavia kirjastoja. Lisäksi kyseistä kirjastoa voidaan käyttää sekä palvelin-, että asiakasohjelmia varten. Kirjaston hyödynnettävyyttä lisää se, että se on saatavissa sekä Scalalle, että Javalle. Serialisointi JSON-muodosta olioiksi ja toisinpäin tapahtui Spray-JSON -kirjaston avulla. Digitransit rajapintaa suositellaan kutsut-

tavaksi GraphQL-kielellä JSON:in sijaan. Tätä varten ei ollut kuitenkaan saatavissa asiakaspuolen kirjastoa Scalalle, joten GraphQL-muotoiset kutsut kirjoitettiin suoraan merkkijonoina.

Kuvassa 12 on esiteltynä sovelluksen sekvenssit pääpiirteittäin. Huomioon otettavaa on, että kaikki kutsut ovat asynkronisia, joten sekvenssikaavioon ei ole merkitty arvojen palauttamista. Ensin haetaan linjan tiedot, kuten ID ja päämäärän nimi. Nämä tiedot yhdistetään toisista Digitransitin REST-endpointeista ID:n avulla haettavien pysäkkien ja kulkuneuvojen sijaintitietojen kanssa.



Kuva 12. Sovelluksen sekvenssit pääpiirteittäin

3.4.1 Valitut kirjastot

Scala-tuen ja Reactive Streams -rajapinnan toteuttamisen, laajan dokumentaation ja yleiskäyttöisyyden ansiosta lupaavin vaihtoehto oli Reactive Extensions. Valintaan vaikutti myös se, että kyseinen kirjasto kattaa myös osittain siihen pohjautuvien Reactor ja Vert.x Rx -kirjastojen käyttöä. Kirjastosta valittiin kuitenkin Java-versio Reactive Streams standardin tuen ansiosta. Lisäksi valikoitui Akka Streams, sillä sen käyttö Akka http –kirjaston kanssa oli välttämätöntä virtojen käsittelyssä. Sovellus oltaisiin voitu toteuttaa kokonaan

Akka Streams -kirjastoa käyttäen, mutta tällöin Reactive Streams -rajapinnan käyttö olisi ollut keinotekoista.

3.4.2 Sovelluksen käyttö

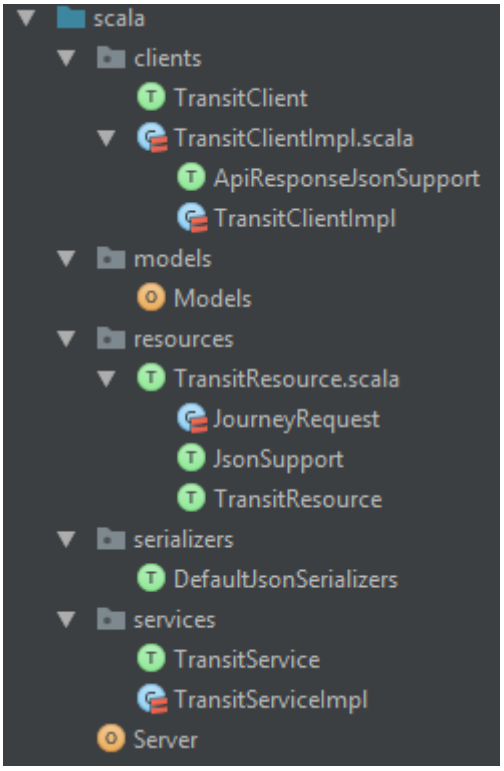
Sovellusta käytetään HTTP-kutsuilla. Sovelluksen ainoa polku sijaitsee osoitteessa /get-journey-info esim. localhost:8080/get-journey-info, jota kutsutaan GET-kutsulla, jonka body-osiossa on haettavan linjan nimi ja kulkuvälineen tyyppi. Kuvassa 13 on curl-ohjelmalla toteutettu kutsu, ja vastaus, kun haetaan raitiovaunulinja 2 tietoja.

```
$ curl -H "Content-Type: application/json" -X GET localhost:8080/get-journey-info -d '{"name": "2", "transitType": "tram"}'

{
  "journey": {
    "id": "HSL:1002",
    "name": "2",
    "destination": "Olympiaterminaali-Kamppi (M)-Töölö-Nordenskiöldinkatu"
  },
  "stops": [
    {
      "name": "Ooppera",
      "coordinates": {
        "latitude": 60.1820157,
        "longitude": 24.9268727
      }
    },
    {
      "name": "Töölöntori",
      "coordinates": {
        "latitude": 60.1795676,
        "longitude": 24.9235216
      }
    }
  ],
  ...
  "vehicles": [
    {
      "coordinates": {
        "latitude": 60.168927,
        "longitude": 24.935604
      }
    },
    {
      "coordinates": {
        "latitude": 60.181551,
        "longitude": 24.927006
      }
    },
    ...
  ]
}
```

Kuva 13. curl-kutsu ja saatu vastaus lyhennettynä

3.4.3 Sovelluksen rakenne

Sovelluksen projektin rakenne on kuvan 14 mukainen. Vertailun kannalta oleelliset osat ovat TransitClient, TransitService ja TransitResource. TransitClient vastaavat tietojen hakemisesta Digitransitin rajapinnoista, TransitService tietojen yhdistämisestä ja TransitResource, REST-endpoint yhdistetyn tiedon palauttamiseen JSON-muotoisena. Näiden lisäksi on Server-luokka REST-endpointin julkaisua varten, Models-luokka, joka sisältää sovelluksen tietotarpeiden mukaiset luokat ja DefaultJsonSerializers, jossa määritetään kyseisten luokkien olioiden serialisointi JSON-muotoon.



Kuva 14. Projektin rakenne

3.5 Sovelluksen versioiden vertailu

Sovelluksen koodista esitellään kohdat, jotka poikkeavat toisistaan, ja ovat tutkimuksen kannalta oleellisia. Esimerkiksi olioiden serialisointi JSON-muotoon on jätetty pois, sillä se on sama kummassakin toteutuksessa. Lisäksi se vie suuren osuuden sovelluksen koodista.

3.5.1 Futuureilla toteutettu versio

Kuvan 15 TransitResource vastaa REST-endpointin määrittämisestä. Siinä määritetään Akka Http:n High-level Server-Side API:a käyttäen sovelluksen polku, joka kutsuu TransitService:n getJourneySummary-metodia, ja deserialisoi futuurista saadun arvon JSON-muotoiseksi sovelluksen käyttäjää varten.

```
trait TransitResource extends JsonSupport {
  implicit val system: ActorSystem
  implicit val materializer: Materializer

  implicit def executor: ExecutionContextExecutor

  val service: TransitService

  val route: Route = {
    path("get-journey-info") {
      (get & entity(as[JourneyRequest])) { journeyRequest =>
        val journeySummary: Future[Option[JourneySummary]] = service.getJourneySummary(
          journeyRequest.name,
          TransitTypes(journeyRequest.transitType)
        )

        onSuccess(journeySummary) { s =>
          complete(s)
        }
      }
    }
  }
}
```

Kuva 15. Futuureilla toteutettu REST-endpoint

Kuvan 16 mukainen TransitService etsii TransitClientia käyttäen linjaa nimen ja kulkuneuvon tyypin perusteella. Mikäli se löytyy, niin linjan tiedot, kuten pysäkkien ja kulkuneuvojen koordinaatit, yhdistetään yhteen JourneySummary-olioon.

```
class TransitServiceImpl(client: TransitClient)(implicit val executionContext: ExecutionContext)
  extends TransitService {

  override def getJourneySummary(journeyName: String, transitType: TransitType): Future[Option[JourneySummary]] = {
    val journeyFuture: Future[Option[Journey]] = client.findDistinctJourney(journeyName, transitType)

    journeyFuture.flatMap {
      case Some(j) => {
        val stops = client.findStops(j.id)
        val vehicles = client.findVehicles(j.id)

        stops.zipWith(vehicles)((s, v) => Some(JourneySummary(j, s, v)))
      }
      case _ => Future(None)
    }
  }
}
```

Kuva 16. Futuureilla toteutettu TransitService

TransitClient-luokan funktiot muistuttavat toisiaan, joten kuvasta 17 on jätetty pois kulkuvälineiden ja pysäkkien hakeminen. Esitetyssä funktiossa kutsutaan Digitransitin rajapintaa, ja määritetään futuuri JSON-muotoisen vastauksen deserialisoimiseksi sovelluksen mukaisiksi olioiksi.

```
override def findDistinctJourney(name: String, transitType: TransitType): Future[Option[Journey]] = {
  val mode = transitType.toString.toUpperCase
  val requestGraphQL =
    s"""
      |{
      |  routes(name: \"$name\", modes: \"$mode\") {
      |    gtfsId
      |    shortName
      |    longName
      |  }
      |}
    """.stripMargin

  val requestData: RequestEntity = ByteString(requestGraphQL)

  val response = Http().singleRequest(
    HttpRequest(
      method = HttpMethods.POST,
      uri = routingEndpoint,
      entity = requestData.withContentType(graphQLContentType)
    )
  )

  response.flatMap(r => Unmarshal(r.entity).to[Seq[Journey]].map(_.find(_.name == name)))
}
```

Kuva 17. Linjojen hakeminen Digitransitin rajapinnasta futuureita käyttäen

3.5.2 Reaktiivisilla kirjastoilla toteutettu versio

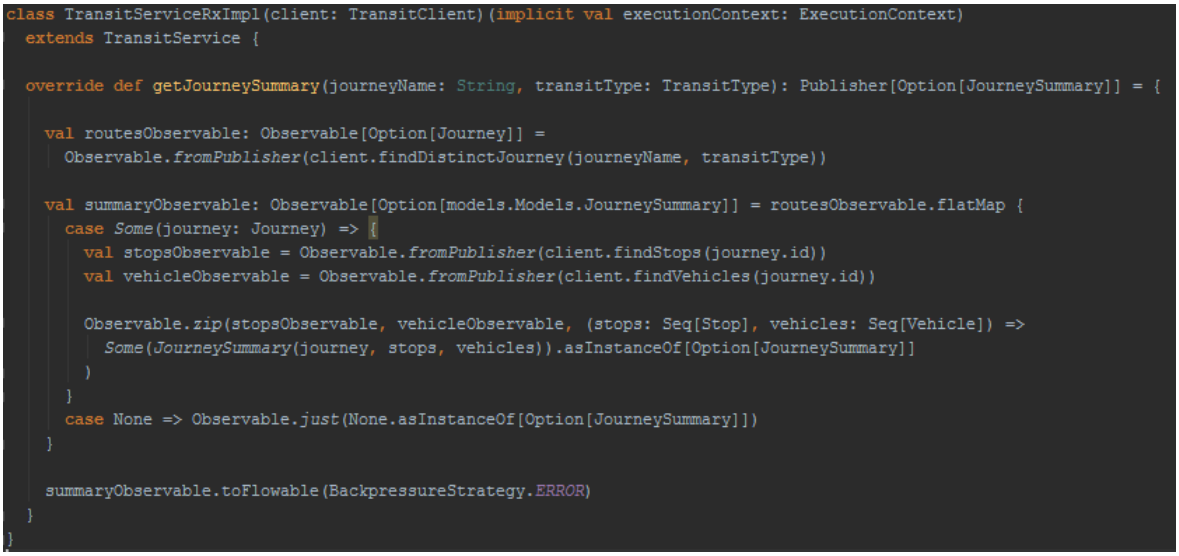
Toisin kuin futuureilla toteutetussa versiossa tieto välitettiin sovelluksen kerrosten välillä Reactive Streams -rajapinnan avulla. Kuten kuvasta 18 nähdään, REST-endpointin määrittämisestä vastaava TransitResource on samankaltainen futuureilla toteutetun version kanssa. Erona siinä on kuitenkin se, että TransitService-luokalta saatu vastaus tulee Reactive Streams mukaisena virtana, joka muutetaan ensin Akka Streams -kirjaston Source-abstraktioksi fromPublisher-metodilla. Source muutetaan lopulta futuuriksi, jonka arvo palautetaan käyttäjälle. Futuurin käyttäminen voitaisiin välttää käyttämällä Akka Http -kirjaston tukemaa JSON-streamausta, mutta tämä tekisi sovelluksen käyttämisestä erilaista kuin futuureilla toteutetussa versiossa.

```
trait TransitResource extends JsonSupport {
  implicit val system: ActorSystem
  implicit val materializer: Materializer

  implicit def executor: ExecutionContextExecutor

  val service: TransitService

  val route: Route = {
    path("get-journey-info") {
      (get & entity(as[JourneyRequest])) { journeyRequest =>
        val p: Publisher[Option[JourneySummary]] = service.getJourneySummary(
          journeyRequest.name,
          TransitTypes(journeyRequest.transitType)
        )

        val result = Source.fromPublisher(p).runWith(Sink.head)

        onSuccess(result) { s =>
          complete(s)
        }
      }
    }
  }
}
```

Kuva 18. Akka Streams:lla toteutettu REST-endpoint

Kuten kuvasta 19 näkyy, RxJavalla toteutetussa versiossa on futuureihin nähden enemmän koodia, mutta suuri osa tulee Reactive Streams:in käytöstä. Lisäksi tässä tapauksessa täytyi tehdä tyyppimuunnos Scalan Option-tyyppiin, mitä ei tarvitsisi tehdä Scala-versiossa. Funktionaalisen ohjelmoinnin piirteiden ansiosta virtojen käsittely tapahtuu kuitenkin samantapaisesti futuurien kanssa, hieman erilaisella syntaksilla tosin. flatMap-funktiolla käsitellään jokaista tarkkailtavan virran elementtiä, ja zip-funktiolla yhdistetään virrat.

RxJavalla pystyy muuttamaan Reactive Streams:n Publisher-rajapinnasta Observable-muotoon kutsumalla Observable-luokan fromPublisher-metodia. Metodilla toFlowable sama hoituu toisinpäin. Tällöin tulee kuitenkin valita, millaista tapaa käytetään toteuttamaan Reactive Streams -standardin mukainen backpressure. Vaihtoehtoja ovat bufferointi, jossa arvoa säilytetään, kunnes virran käsittelijä kuluttaa sen, arvojen pudottaminen, virheen heittäminen, vanhojen arvojen ylikirjoittaminen ja backpressuresta huolehtimisen jättäminen virran käsittelijälle.



```
class TransitServiceRxImpl(client: TransitClient)(implicit val executionContext: ExecutionContext)
  extends TransitService {

  override def getJourneySummary(journeyName: String, transitType: TransitType): Publisher[Option[JourneySummary]] = {

    val routesObservable: Observable[Option[Journey]] =
      Observable.fromPublisher(client.findDistinctJourney(journeyName, transitType))

    val summaryObservable: Observable[Option[models.Models.JourneySummary]] = routesObservable.flatMap {
      case Some(journey: Journey) => {
        val stopsObservable = Observable.fromPublisher(client.findStops(journey.id))
        val vehicleObservable = Observable.fromPublisher(client.findVehicles(journey.id))

        Observable.zip(stopsObservable, vehicleObservable, (stops: Seq[Stop], vehicles: Seq[Vehicle]) =>
          Some(JourneySummary(journey, stops, vehicles)).asInstanceOf[Option[JourneySummary]]
        )
      }
      case None => Observable.just(None.asInstanceOf[Option[JourneySummary]])
    }

    summaryObservable.toFlowable(BackpressureStrategy.ERROR)
  }
}
```

Kuva 19. RxJavalla toteutettu TransitService

Kuvasta 20 voidaan nähdä, miten virtojen käsittely Akka Streamsilla tapahtuu. Ensin määritetään virran lähde, jolle määritetään transformaatio. Lähde ja sen muuntaminen suoritetaan, ja saatu tulos muunnetaan Reactive Streams:in mukaiseksi Sink-luokan asPublisher-metodilla. Kyseinen metodin fanout-parametri määrittää käytettävän backpressuren. Akka Streams backpressuren määrittäminen poikkeaa RxJavasta. Backpressure vaihtoehtoja on kaksi, toinen on tarkoitettu yhdelle käsittelijälle, ja toinen useammalle. Monelle käsittelijälle tarkoitettu backpressure hidastaa elementtien julkaisua hitaimman käsittelijän tasolle. Yhdelle tarkoitettu puolestaan heittää virheen, mikäli virralla on useampi kuin yksi käsittelijä.

```
override def findDistinctJourney(name: String, transitType: TransitType): Publisher[Option[Journey]] = {
  val mode = transitType.toString.toUpperCase
  val requestGraphQL =
    s"""
      |{
      |  routes(name: \"$name\", modes: \"$mode\") {
      |    gtfsId
      |    shortName
      |    longName
      |  }
      |}
    """.stripMargin

  val requestData: RequestEntity = ByteString(requestGraphQL)

  val source: Source[HttpRequest, NotUsed] = Source.single(
    HttpRequest(
      method = HttpMethods.POST,
      uri = routingEndpoint,
      entity = requestData.withContentType(graphQLContentType)
    )
  )

  val flow = Http().outgoingConnectionHttps(baseUri).mapAsync(1) { r =>
    Unmarshal(r.entity).to[Seq[Journey]].map(_.find(_.name == name))
  }

  source.via(flow).runWith(Sink.asPublisher(fanout = true))
}
```

Kuva 20. Linjojen hakeminen Digitransitin rajapinnasta Akka Streamsia käyttäen

3.6 Tulokset ja yhteenveto

IntelliJ Idea:n Statistic-pluginilla laskettuna futuureilla toteutetussa versiossa Scala-tiedostojen koodirivien lukumäärä oli yhteensä 388 riviä. Rxjavaa ja Akka Streamsia käyttävässä versiossa rivien määrä oli 412. Reaktiivisilla kirjastoilla toteutettu kirjasto käytti myös Reactive Streams-standardia. Tällä tavalla toteutetun sovelluksen koodin luettavuus ei parantunut reaktiivisten kirjastojen avulla.

Kirjastoja lähemmin tarkasteltaessa vahvistui käsitys siitä, että reaktiivisen ohjelmoinnin kirjastot Scalalle ovat pääsääntöisesti luokittelematonta reaktiivista ohjelmointia. Lisäksi muissa paitsi Vert.x Sync -kirjastossa muutosten propagointi toteutetaan funktionaalisen komposition avulla. Riippuvuudet, joiden välillä muutokset propagoidaan, määritetään pääosin funktionaalisesti zip- tai flatMap-funktioiden avulla, ja tieto muutoksista levitetään tämän luodun graafin sisällä. Tutkimuksessa käytettyjen kriteerien mukaisesti kaupallisesti merkittäviä funktionaalisia reaktiivisia kirjastoja ei kuitenkaan löytynyt. Tästä ei silti voida vetää suoraan johtopäätöstä, että niitä ei ole. Taulukossa 2 on yhteenveto käsiteltyjen kirjastojen reaktiivisen ohjelmoinnin piirteistä, ja mihin alalajiin ne kuuluvat.

Taulukko 2. Yhteenveto kirjastojen reaktiivisen ohjelmoinnin piirteistä

Kirjasto	**Automaattinen muutosten propagointi**	**Jatkuvaan aikaan pohjautuva abstraktio**	**Asynkroninen**	**Reaktiivisen ohjelmoinnin alalaji**
Akka Streams	Kyllä	Ei	Kyllä	Muu/luokittelematon
Apache Spark Streaming	Kyllä	Ei	Kyllä	Muu/luokittelematon
Reactor	Kyllä	Ei	Kyllä	Muu/luokittelematon
RxJava/RxScala	Kyllä	Ei	Kyllä	Muu/luokittelematon
Sodium	Kyllä	Kyllä	Kyllä	FRP
Vert.x Rx	Kyllä	Ei	Kyllä	Muu/luokittelematon
Vert.x Sync	Kyllä	Ei	Kyllä	Muu/luokittelematon

4 Pohdinta

Reaktiivisen ohjelmoinnin kirjastoja tutkittaessa ongelmana oli selvittää mitä reaktiivinen ohjelmointi ylipäätään on. Funktionaalisella reaktiivisella ohjelmoinnilla on selvä mateemaattinen määritelmä, mutta haasteita aiheutti reaktiivisen ohjelmoinnin käsite. Absoluuttista määritelmää ei löytynyt, ja kirjallisuus perustuu pitkälti tulkintoihin ja FRP-toteutuksista koottuihin havaintoihin. Teorian ja empiirisen osan perusteella tehdään kuitenkin seuraavanlainen tulkinta: Reaktiivisessa ohjelmoinnissa keskeisimmät piirteet ovat deklaratiivisuus ja tapahtumapohjaisuus. Tapahtumien käsittely abstraktoidaan reaktioihin, joita voidaan yhdistää graafeiksi, joiden sisällä tieto muutoksesta välitetään automaattisesti.

Mainintaa asynkronisuudesta ei löytynyt varhaisista reaktiivista ohjelmointia käsittelevistä lähteistä. Se kuitenkin soveltuu ilmeisen hyvin reaktiiviseen ohjelmointiin, ja onkin usein keskeisenä asiana reaktiivisissa Scala-/Java-kirjastoissa, mikä näkyi käsitellyistä kirjastoista. Deklaratiivisuus mahdollistaa asynkronisuuden toteuttamisen niin, että se tapahtuu toteutusta käyttävän ohjelmoijan näkökulmasta automaattisesti. Scalan futuurit voidaan nähdä reaktiiviseen ohjelmointiin kuuluvana, vaikka ne ovat nimenomaan asynkroniseen ohjelmointiin tarkoitettuja. Niillä voidaan toteuttaa reaktiivisen ohjelmoinnin piirteitä, kuten muutosten propagointi, mikäli niitä komposoidaan funktionaalisesti.

Tutkimuksessa käsitellyt reaktiivisen ohjelmoinnin kirjastot vahvistivat Bonérin ja Klangin (2016) näkemyksessä siitä, että suurin osa reaktiivisen ohjelmoinnin kirjastoista ovat muuta kuin funktionaalista reaktiivista ohjelmointia. Tutkimukseen valikoitui yksi FRP-kirjasto – Sodium, joka valittiin mukaan nimenomaan FRP-piirteiden vuoksi. Kirjastoja valittaessa otettiin huomioon myös muita tutkimuksessa käsittelemättömiä FRP-kirjastoja, kuten Scala.React, Scala.Rx ja Scala.FRP. Näissä oli kuitenkin ongelmina mm. se, ettei niitä oltu joko päivitetty moneen vuoteen tai niistä puuttui joitakin keskeisiä FRP-piirteitä, kuten käytökset. Tutkimuksessa käsiteltyjä kirjastoja yhdistäviä reaktiivisen ohjelmoinnin piirteitä olivat muutosten propagointi ja asynkronisuus. Lisäksi eräänä yhdistävänä tekijänä oli tapahtumien kuvaaminen virtoina. Poikkeuksena tähän oli kuitenkin Vert.x Sync, jossa pystytään käyttämään myös dataflow-muuttujia.

Esimerkkisovelluksesta saatujen tulosten mukaan reaktiivisen ohjelmoinnin kirjastojen käyttö kasvatti koodirivien määrää yksinkertaisessa sovelluksessa, ja subjektiivisesti katsottuna vaikeutti koodin luettavuutta. Tulokset eivät ole yllättäviä, sillä virtoihin perustuvat ratkaisut soveltuvat parhaiten suuren tietomäärän käsittelyyn. Jatkotutkimusta voisikin tehdä reaktiivisen ohjelmoinnin kirjastojen soveltuvuudesta toisenlaisiin käyttökohteisiin.

Kirjastojen soveltuvuus erilaisiin projekteihin on tapauskohtaista. Akka Streams voi olla sopiva vaihtoehto pohjaksi, jos toteutetaan kirjastoa Scalalle, varsinkin jos käytetään muita Akka-tuoteperheen kirjastoja. Konkreettinen esimerkki voisi olla tietokantakirjasto. Jos kuitenkin haetaan mahdollisimman yleiskäyttöistä reaktiivista kirjastoa, niin Reactive Extensions voi olla sopiva valinta. Sillä on laaja dokumentaatio, ja useita merkittäviä käyttäjiä, mikä turvaa sen ylläpidon lähitulevaisuudessa. Lisäksi kirjasto tukee reaktiivisia kirjastoja yhdistävää Reactive Streams -standardia, minkä merkitys saattaa kasvaa Java 9 -standardin myötä. RxScala ei kuitenkaan vielä toteuta standardia, joten toistaiseksi RxJava voi olla sopivampi valinta myös Scalaa käytettäessä. Mikäli funktionaalista ohjelmointia ei tahdota hyödyntää, niin soveltuvin vaihtoehto voi olla Vert.x Sync, joka muistuttaa käytöltään enemmän perinteistä olio-ohjelmointia. Sen dataflow-muuttujien käyttö on samankaltaista kuin Javan futuureilla. Kirjasto ei ole kuitenkaan suositeltava Scalalle, joka on nimenomaisesti suunniteltu funktionaalinen ohjelmointi mielessä.

Jos mietitään millainen kirjastokokonaisuus saattaisi olla hyvä Scalalla backend-puolen sovellusta varten, niin valintoina voisi olla Akka Http palvelinkutsujen ja vastausten hallintaan, RxJava/RxScala logiikkaa varten, ja Slick tietokantaa varten. Nämä toteuttavat Reactive Streams-standardin, joten niiden yhdistäminen toisiinsa on vaivatonta. Toinen vaihtoehto olisi käyttää Scalalla suosittua Play-ohjelmistokehystä, jossa on kokeellisessa tilassa kirjasto Reactive Streams -integraatioita varten.

Esimerkkisovelluksesta saatuja tuloksia voidaan pitää suuntaa antavina. Koodirivien määrä riippuu paljon ohjelmoijasta, ja käytettävistä tekniikoista. Jos sama oltaisiin toteutettu takaisinkutsuilla futuurien sijaan koodirivien määrä saattaisi kasvaa huomattavasti. Lisäksi palvelin- ja asiakaskirjastona käytetty Akka Http toimi ehkä reaktiivisen ohjelmoinnin eduksi. Jos oltaisiin käytetty suoraan Scala/Java Servlettejä, ero natiivin ja kirjastoilla toteutetun version välillä saattaisi kasvaa entisestään. Tällaisessa tapauksessa futuureilla toteutettu versio olisi ollut oletettavasti tiiviimpi.

Alun perin suunnitelmana oli rajata työtä käsittelemään funktionaalista reaktiivista ohjelmointia. Työn edetessä kävi kuitenkin ilmi, ettei se ole merkittävä Scalan näkökulmasta. Tämän olisi voinut välttää, mikäli reaktiivinen ohjelmointi olisi ollut täysin vieras käsite alun perin, ja ennakkokäsitykset eivät olisi ohjanneet työskentelyä. Bonérin ja Klangin (2016.) reaktiivisten käsitteiden sekaannusten käsitteleminen kuitenkin auttoi opinnäytetyötä eteenpäin. Alkuperäisenä ajatuksena oli pitäytyä mahdollisimman käytännönläheisessä näkökulmassa. Tämä ei kuitenkaan ollut mahdollista ilman selvää näkemystä käsitteistä.

Kaiken kaikkiaan opinnäytetyöstä saatu ammatillinen kehittyminen oli merkittävää. Reaktiiviseen ohjelmointiin tutustumisen lisäksi opinnäytetyö mahdollisti syvemmän perehtymisen Scalaan ja funktionaaliseen ohjelmointiin. Erityisesti FRP:tä käsitteleviä lähteitä lukiessa tarkentui mihin funktionaalinen ohjelmointi perustuu. Aiempi tieto oli peräisin pääasiassa käytännön kokemuksista. Lisäksi tutkimus antoi syyn perehtyä aiemmin kiinnostusta herättäneisiin teknologioihin, kuten Reactive Extensions -kirjastoihin. Kirjastoihin perehtyminen tarkensi myös käsitystä siitä, mihin ne ja reaktiivinen ohjelmointi ylipäätään soveltuvat parhaiten. Tutkimus selvensi myös sen, että Slick-tietokantakirjaston reaktiivisuudella tarkoitetaan reaktiivista ohjelmointia, mutta kirjasto ei ole kuitenkaan funktionaalisista piirteistään huolimatta FRP:n mukainen. Työelämässä kohdatut kirjaston käytön muuttumisesta johtuneet ongelmat eivät liittyneet suoraan reaktiiviseen ohjelmoimiseen, vaan ne liittyivät asynkronisen ohjelmoinnin kirjaston liittämisestä synkroniseen koodiin. Tästä saisi aiheen jatkotutkimukselle, vaikka se ei liity suoraan reaktiiviseen ohjelmointiin.

Lähteet

Bainomugisha, E & Carreton, A & van Cutsem, T & Mostinckx, S & de Meuter, W. 2013. A Survey on Reactive Programming. ACM Computing Surveys, 45, 4. ACM. Luettavissa: http://dl.acm.org/citation.cfm?id=2501666. Luettu: 10.11.2016.

Blackheath, S & Jones, A. 2016. Functional Reactive Programming. Manning.

Bonér, J & Farley, D & Kuhn, R & and Thompson, M. 2014. The Reactive Manifesto. Luettavissa: http://www.reactivemanifesto.org/. Luettu: 5.1.2017.

Bonér, J & Klang, V. 2016. Reactive Programming versus Reactive System. Lightbend. Luettavissa: https://www.lightbend.com/reactive-programming-versus-reactive-systems. Luettu: 20.1.2017.

Boussinot, F. 1991. Reactive C: An Extension of C to Program Reactive Systems. Software—Practice & Experience, 21, 4, s. 401-428.
Luettavissa: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.2178&rep=rep1&type=pdf. Luettu: 9.1.2017.

Brisbin, J. 2013. Reactor – a foundation for asynchronous applications on the JVM. Luettavissa: https://spring.io/blog/2013/05/13/reactor-a-foundation-for-asynchronous-applications-on-the-jvm. Luettu: 11.2.2017.

Edwards, J. 2009. Coherent Reaction. Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, s. 925-932. ACM. Luettavissa: http://www.subtext-lang.org/Onward09.pdf. Luettu: 30.3.2017.

Elliott, C. 2009. Push-Pull Functional Reactive Programming. OOPSLA ’09 Proceedings of the 2nd ACM SIGPLAN symposium on Haskell, s. 25-36. ACM. Luettavissa: http://conal.net/papers/push-pull-frp/push-pull-frp.pdf. Luettu: 26.1.2017.

Elliott, C. 13.6.2015. The essence and origins of FRP. Konferenssiesitys. BayHac 2015. Mountain View, CA, Yhdysvallat.
Katsottavissa: https://begriffs.com/posts/2015-07-22-essence-of-frp.html. Katsottu: 26.1.2017.

Elliott, C. 2015. The essence and origins of FRP. Luettavissa: http://conal.net/talks/essence-and-origins-of-frp-bayhac-2015.pdf. Luettu: 26.1.2017.

EPFL. 2016. Stream. Luettavissa: http://www.scala-lang.org/api/2.12.0/scala/collection/immutable/Stream.html. Luettu 14.2.2017

Haller, P & Prokopec, A & Miller, H & Klang, V & Kuhn, V & Jovanovic, Vojin. 2015. Futures and Promises. Luettavissa: http://docs.scala-lang.org/overviews/core/futures.html. Luettu: 8.2.2017.

Hudak, P & Courtney, A & Nilsson, H & Peterson, J. 2003. Arrows, Robots, and Functional Reactive Programming. Luettavissa: http://haskell.cs.yale.edu/wp-content/uploads/2011/02/oxford02.pdf. Luettu: 24.1.2017.

Gamma, E & Helm, R & Johnson, R & Vlissides, J. 1995. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley.

Lightbend inc. 2016. Akka. Luettavissa: http://akka.io/. Luettu: 1.3.2017.

Kapadia, S. 2016. Simply explained: Akka Streams Backpressure. Luettavissa: http://chariotsolutions.com/blog/post/simply-explained-akka-streams-backpressure/. Luettu: 25.3.2017.

Maier, I & Odersky, M. 2012. Deprecating the Observer Pattern with Scala.React. Luettavissa: https://infoscience.epfl.ch/record/176887/files/DeprecatingObservers2012.pdf. Luettu: 17.1.2017.

Malawski, Konrad. 2015. Akka Streams & Http 1.0 Released!. Luettavissa: Akka Streams & Http 1.0 Released!. Luettu: 12.2.2017.

Nilsson, H & Courtney, A & Peterson, J. 2002. Functional Reactive Programming, Continued. Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, s.51-64. ACM. Luettavissa: http://haskell.cs.yale.edu/wp-content/uploads/2011/02/workshop-02.pdf. Luettu: 25.3.2017.

Odersky, M & Altherr, P & Cremet, V & Emir, B & Maneth, S & Micheloud, S & Mihaylov, N & Schinz, M & Stenman, E & Zenger, M. 2004. An Overview of the Scala Programming

Language. Luettavissa: https://infoscience.epfl.ch/record/52656/files/ScalaOverview.pdf. Luettu: 18.1.2016.

Oracle. 2017. Flow. Luettavissa: http://download.java.net/java/jdk9/docs/api/java/util/concurrent/Flow.html. Luettu 9.2.2017.

Reactive Streams Special Interest Group. 2015. Reactive Streams 1.0.0 is here!. Luettavissa: http://www.reactive-streams.org/announce-1.0.0. Luettu: 9.2.2017.

Reactive Streams Special Interest Group. 2016. Reactive Streams. Luettavissa: https://github.com/reactive-streams/reactive-streams-jvm/blob/master/README.md. Luettu: 9.2.2017.

ReactiveX. 2017a. ReactiveX. Luettavissa: http://reactivex.io. Luettu: 24.3.2017.

ReactiveX. 2017b. Observer. Luettavissa: http://reactivex.io/documentation/observable.html. Luettu: 24.3.2017.

Schmidt, D. 1995. Reactor – An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events. Luettavissa: http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf. Luettu: 24.3.2017.

Van-Roy, P & Haridi, S. 2004. Concepts, Techniques, and Models of Computer Programming. The MIT Press.

Wan, Z & Hudak, P. 2000. Functional Reactive Programming from First Principles. PLDI '00 Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, s. 241-252. ACM. Luettavissa: http://dl.acm.org/citation.cfm?id=349331&CFID=898056549&CFTOKEN=65019158. Luettu: 7.11.2016.

Watt, D. 2004. Programming Language Design Concepts. Wiley.